

2 VARIABLES AND CONSTANTS

Programs can process data in a variety of ways. They may, for example, perform calculations with numbers, or save and recall *strings* of text (such as names and phone numbers in a data file).

In all cases, your program must be able to handle *values* - different types of numbers, strings, and so on.

In OPL, there are two ways of handling values: *variables* and *constants*. Constants are fixed values (which may be named on the Series 5). Variables are used to store values which may change - for example, a variable called x may start with the value 3 but later take the value 7.

be exceed this and take any value in the full range of 64-bit floating point numbers (see the Series 5 section above) .

Text

For text - Are you sure?, 54th, etc. - use a **string variable**. (Pieces of text are called *strings* in OPL.) String variables have a \$ symbol on the end - for example, name\$.

To declare a string variable, you **must** follow the \$ symbol with the maximum length of string you want the variable to handle in brackets. So if you want to store names up to 15 characters long in the variable name\$, declare it like this: LOCAL name\$ (15) Strings cannot be longer than 255 characters.

Array variables

You may want a group of variables, for example to store lists of values. Instead of having to declare separate variables a, b, c, d and e, you can declare *array* variables a (1) to a (5) in one go like this:

LOCAL a% (5) (array of integer variables)

LOCAL a (5) (array of floating-point variables)

LOCAL a\$ (5, 8) (array of string variables)

or

LOCAL a& (5) (array of long integers)

The number in brackets is the number of elements in the array. So LOCAL a% (5) creates five integer variables: a% (1), a% (2), a% (3), a% (4) and a% (5) .

With strings, the second number in the brackets specifies the maximum length of the strings. All the elements in the string array have the same capacity - for example, LOCAL ID\$ (5, 10) allocates memory space for five strings, each up to 10 characters in length.

OPL does not support two-dimensional arrays.

Initial values

All numeric variables have zero as their initial value. String variables have a string with no characters in it (a *null* or *empty* string). Every element in an array variable is also initialised in the appropriate way.

Choosing descriptive names

To make it easier to write your programs, and understand them when you read through them at a later date, give your main variables names which describe the values they hold. For example, in a procedure which calculates fuel efficiency, you might use variables named *speed* and *distance*.

All variable names:

- May be entered in any combination of upper and lower case. sPeEd and SpEEd would be considered the same name.
- Must not use any of the names of keywords, as listed in the ‘Alphabetic Listing’ chapter - if you use these you will see a ‘Declaration error’ message when you translate your module.

Other constraints are machine dependent:

↑

- May be up to 32 characters long
- **Must** start with either an underscore (_) or an alphabetic character, but after that may use any combination of numbers, letters and the underscore character.

↓

- May be up to 8 characters long
- **Must** start with an alphabetic character, but after that may use any combination of numbers and letters

The \$, & and % symbols are included in the 32 (or 8) characters allowed in variable names, so V2345678901234567890123456789012% is too long to be a valid variable name, but V234567890123456789012345678901% is acceptable (on the Series 5).

EXAMPLES

- LOCAL clients\$(12), z&(3) declares one string variable, clients\$, of capacity 12 characters, and one long integer array variable containing three elements, z&(1), z&(2) and z&(3)
- LOCAL AGE%, B5\$(10), i declares one integer variable, AGE%, one string variable, B5\$, of capacity 10 characters, and one floating-point variable, i
- LOCAL profit93 declares one floating-point variable, profit93
- LOCAL x, MAN6\$(4, 7) declares one floating-point variable, x, and one string array variable, man6\$, containing four elements, man6\$(1), man6\$(2), man6\$(3) and man6\$(4), each of capacity 7 characters

For efficiency

- Integer variables use less memory than long integer variables, and both use less than floating-point.
- Integer variables are processed faster than floating-point.

GIVING VALUES TO VARIABLES

Assigning values

You can *assign* a value to a variable directly, like this:

```
x=5
```

```
y=10
```

This procedure adds two numbers together:

```
PROC add:
    LOCAL x%, y%, z%
    x%=569
    y%=203
    z%=x%+y%
    PRINT z%
    GET
ENDP
```

add: is the procedure name.

The LOCAL statement defines three variables x%, y% and z%, all initially with the value 0. PRINT displays the value of z% on the screen. You can display the value of any variable like this.

PROC and ENDP define the beginning and end of the procedure as you saw in the previous chapter.

Assigning values to string variables

String variables can be assigned text values like this:

```
a$="some text"
```

The text you use must be enclosed in double quote characters.

Assigning values to an array variable

If you declare `a%(4)`, assign values to each of the elements in the array like this: `a%(1)=56`, `a%(2)=345` and so on. Similarly for the other variable types: `a(1)=.0346`, `a&(3)=355440`, `a$(10)="name"`.

Arithmetic operations

You can use these *operators*:

+	plus
-	minus or make negative
/	divide
*	multiply
**	raise to a power
%	percentage

Operators have the same precedence as in the Calc application. For example, `3+51.3/8` is treated as `3+(51.3/8)`, not `(3+51.3)/8`. For more information on operators and precedence, see Appendix B.

Values from functions

There are two kinds of keyword - *commands* and *functions*:

- A command is just a straightforward instruction to OPL to do some particular thing. PRINT and PAUSE, for example, are commands.
- A function is just like a command but it also *returns* a value which you can then use.

GET is, in fact, a function; it waits for you to press a key on the keyboard, and then returns a value which identifies the key which was pressed. (In previous example programs, the value returned by GET was ignored, as GET was being used to provide a pause while you read the screen. This is a common use of the GET function.)

The number returned by GET will always be a small whole number, so you might store it away in an integer variable, like this:

```
a%=GET
```

There is more about the GET function later in this chapter.

Expressions

You can assign a value to a variable with an *expression* - that is, a combination of numbers, variables, and functions. For example:

`z=x+y/2` gives the z the value of x plus the value of y/2.

`z=x*y+34.78` gives z the value of x times y, plus 34.78.

`z=x+COS(y)` gives z the value of x plus the cosine of y.

COS is another OPL function. Unlike the GET function, COS requires a value or variable to work with. As you can see, you put this in brackets, after the function name. Values you give to functions in this way are called *arguments* to the function. There is more information about arguments in the next chapter.

All of the above are *operations* using the variables x and y - assigning the result to z and not actually affecting the value of x or y.

The ways you can change the values of variables fall into these groups:

- Arithmetic operations, such as multiplication or addition - for example $z = \text{sales} + \text{costs}$ or $z = y\% * (4 - x\%)$
- Using one of the OPL functions, for example $z = \text{SIN}(\text{PI} / 6)$

or

- Using certain keywords like INPUT or EDIT which wait for you to type in values from the keyboard.

Self reference

In expressions, variables can refer to themselves. For example:

$z\% = z\% + 1$ (make the value of $z\%$ one greater than its current value)

$x\% = x\% / 4 + y$ (make the value of $x\%$ a quarter of its current value, plus the value of y)

Constants

In an OPL program, numbers (and strings in quote marks) are sometimes called *constants*. In practice, you will use constants without thinking about them. For example:

$x = 0.32$

$x\% = 569$

$x\& = 32768$

$x\$ = \text{"string"}$

$x(1) = 4.87$

OPL can also represent *hexadecimal* constants. Integers specified in hexadecimal must be preceded by a \$ and long integers by a &. For example, \$f or &8000000. This is explained under the HEX\$ entry in the 'Alphabetic Listing' chapter.

Exponential notation may be useful for very large or very small numbers. Use E (capital or lower case) to mean "times ten to the power of" - for example, 3.14E7 is 3.14×10^7 (31400000), while 1E-9 is 1×10^{-9} (0.000000001).

I The CONST command may be used to declare *constants*. This makes it possible to assign a name to a constant value so it may be used throughout the module. This has the advantage of making it possible to change just one statement rather than many to change the value of a single constant. See the 'Calling Procedures' chapter for more details of how to do this.

Problems with integers

When calculating an expression, OPL uses the simplest arithmetic possible for the numbers involved. If all of the numbers are integers, integer arithmetic is used; if one is outside integer range, but within long integer range, then long integer arithmetic is used; if any of the numbers are not whole numbers, or are outside long integer range, floating-point arithmetic is used.

This has the benefit of maximising speed, but you must beware of calculations going out of the range of the type of arithmetic used. For example, in $X = 200 * 300$ both 200 and 300 are integers, so integer arithmetic is used for speed (even though X is a floating-point variable). However, the result, 60000, cannot be calculated because it is outside integer range (32767 to -32768), so an 'Overflow' error is produced.

You can get around this by using the INT function, which turns an integer into a long integer, without changing its value. If you rewrite the previous example as $X = \text{INT}(200) * 300$, OPL has to use long integer arithmetic, and can therefore give the correct result (60000). (If you understand hexadecimal numbers, you can instead write one of the numbers as a hexadecimal long integer, e.g. 200 would become &C8.)

Integer arithmetic uses whole numbers only. For example, if $y\%$ is 7 and $x\%$ is 4, $y\%/x\%$ gives 1. However, you can use the INTF function to convert an integer or long integer into a floating-point number, forcing floating-point arithmetic to be used for example, $\text{INTF}(y\%)/x\%$ gives 1.75. **This rule applies to each part of an expression** - e.g. $1.0+2/4$ works out as $1.0+0$ ($=1.0$), while $1+2.0/4$ works out as $1+0.5$ ($=1.5$).

If one of the integers in an all-integer calculation is a constant, you can instead write it as a floating-point number. $7/4$ gives 1, but $7/4.0$ gives 1.75.

Operations on strings

If $a\%$ is "down" and $b\%$ is "wind", then the statement $c\%=a\%+b\%$ means $c\%$ becomes "downwind".

Alternatively, you could give $c\%$ the same value with the statement $c\%="down"+"wind"$.

When adding strings together, the result must not be longer than the maximum length you declared e.g. if you declared `LOCAL a$(5)` then $a\%="first"+"second"$ would cause a 'String is too long' error to be displayed.

Most operators do not work on strings. To cut up strings, use string functions like MID\$, LEFT\$ and RIGHT\$, explained in the 'Alphabetic Listing' chapter. You need them to extract even a single character you **cannot**, for example, refer to the fourth character in $a\%(7)$ as $a\%(4)$.

DISPLAYING VARIABLES

PRINT is one of the most useful OPL commands. Use it to display any combination of text messages and the values of variables.

Where the cursor goes after a PRINT

In general, each PRINT statement ends by moving to a new line. For example:

```
A%=127 :PRINT "A% is"
PRINT a%
```

would display as

```
A% is
127
```

You can stop a PRINT statement from moving to a new line by ending it with a semicolon. For example:

```
A%=127 :PRINT "A% is";
PRINT a%
```

would display as

```
A% is127
```

If you end a PRINT statement with a comma, it stays on the same line, but displays an extra space. For example:

```
A%=127 :PRINT "A% is",
PRINT a%
```

would display as

```
A% is 127
```


Displaying a list of things

You can use commas or semicolons to separate things to be displayed on one line, instead of using one PRINT statement for each. They have the same effect as before:

```
A%=127 :PRINT "A% is",a%
```

would display as

```
A% is 127
```

while

```
user$="Fred"
```

```
PRINT "Hello",user$;"!"
```

would display as

```
Hello Fred!
```

Displaying the quote character

Each string you use with PRINT must start and end with a quote character. Inside the string to display, you can represent the quote character itself by entering it twice. So PRINT "Press "" key" displays as Press " key, while PRINT """" displays a single quote character.

VALUES FROM THE KEYBOARD

If you want a program to be reusable, it often needs to be able to accept different sets of information each time you use it. You can do this with the INPUT command, which takes numbers and text typed in at the keyboard and stores them in variables.

For example, this simple procedure converts from Pounds Sterling to Deutschmarks. It asks you to type in two numbers - the number of Pounds Sterling, and the current exchange rate. You can edit as you type the numbers - the Delete key, for example, deletes characters, and Esc clears everything you've typed. Press Enter when you've finished each number. The values are assigned to the variables `pounds` and `rate`, and the result of the conversion is then displayed:

```
PROC exch:
```

```
LOCAL pounds,rate
```

```
AT 1,4
```

```
PRINT "How many Pounds Sterling?",
```

```
INPUT pounds :REM value from keyboard
```

```
PRINT "Exchange rate (DM to £1)?",
```

```
INPUT rate :REM value from keyboard
```

```
PRINT "=",pounds*rate,"Deutschmarks"
```

```
GET
```

```
ENDP
```

Here PRINT is used to show messages (often called *prompts*) before the two INPUT commands, to say what information needs to be typed in. In both cases the PRINT command ends in a comma, which displays a single space, and keeps the cursor position on the same line. Without the commas, the numbers you type to the INPUT commands would appear on the line below.

The value entered to an INPUT command must be of the appropriate kind for the variable which INPUT is setting. If you enter the wrong type (for example, if you enter the string `three` for the floating-point variable `rate`), INPUT will show a ? prompt, and wait for you to enter another value.

When using INPUT with a numeric variable (integer, long integer or floating-point), you can enter any number within the range of that type of variable. Note that if you enter a non-whole number as the

value for an integer variable, it will take only the whole number part (so e.g. if you enter 12.75 for an integer variable, it will be set to 12).

Comments

The REM command lets you add comments to a program to help explain how it works. Begin the comment with the word REM (short for 'remark'). Everything after the REM command is ignored.

If you put a REM command on the end of a line, the colon you would normally put before it is optional. For example, you could use either of these:

```
CLS :REM Clears the screen
```

or

```
CLS REM Clears the screen
```

AT command

This positions the cursor or your message at the co-ordinates you specify. Use the command like this:

```
AT column%,row%
```

where column% and row% give the character position to use.

AT 1,1 positions the cursor to the top left corner.

Single keypresses

In addition to using INPUT to ask for values, your program can ask for single keypresses. Use one of these functions:

- GET waits for a keypress and returns the key pressed.
- KEY returns a key if any was pressed, but doesn't wait for one.

Every separate letter, number or symbol has a number which represents it, called a *character code*. The full list of character codes - the *character set* - for the Series 5 may be found in Appendix D and for the Series 3c is included as an appendix to the User Guide. GET and KEY return the character code of the key pressed for example, if A were pressed, these functions would return the value 65. KEY returns 0 if no key was pressed.

KEY\$ and GET\$ work in the same way as KEY and GET, except that they return the key pressed as a string, not as a character code:

- GET\$ waits for a keypress and returns the key pressed, as a string.
- KEY\$ returns a key if any was pressed, but doesn't wait for one. KEY\$ returns a null string if no key was pressed.

Unlike INPUT, these functions do not display the key pressed on the screen, and do not wait for you to press Enter.

Example using GET\$

```
PROC kchar:
    LOCAL k$(1)
    PRINT "Press a key, A-Z:"
    k$=GET$
    PRINT "You pressed",k$
    PAUSE 60
ENDP
```

Single keypresses are often useful for making decisions. A program might, for example, offer a set of choices which you choose from by typing the word's first letter, like this:

Add (A) Erase (E) or Copy (C) ?

Or it might ask for confirmation of a decision, by displaying a YES or NO? message and waiting until Y or N is pressed.

See the 'Loops and Branches' chapter for details of how to identify which key is pressed.

Modifier keys

If you need to check for the Shift, Control, Psion (on the Series 3c and Siena only) Fn (Series 5 only) keys and/or Caps Lock being used, see the description of the KMOD function, in the 'Alphabetic Listing' chapter.

SUMMARY

Declare variables with one or more LOCAL statements in the line after PROC:

- **Integer** variables - for example `year%`
- **Floating-point** variables - for example `price`
- **String** variables - for example `name$(12)` where the maximum length is given in the brackets
- **Long integer** variables - for example `profit&`

Variables will be floating-point unless you add a symbol to the end of the variable name.

- **Array** variables - for example `prices%(4)` or `clients$(5,12)` where the first number inside the brackets specifies the number of elements, and the second number in the brackets, in the case of string arrays, specifies the maximum length.

All identifiers may have a maximum length of 32 characters (8 on the Series 3c).

Assign values to variables:

- Expressions - for example `x=5.5/y`, `profit=x-y`
- INPUT command - for example `INPUT a$`
- 'Add' strings - for example `a$="MR"+names$`

REM allows you to add comments to a program.

AT positions the cursor.

GET and KEY return the key pressed as a character code.

GET\$ and KEY\$ return the key pressed as a single-character string.

GET and GET\$ wait until a key is pressed, KEY and KEY\$ do not.